Tarea 2:

Multiplicación óptima de Matrices

Nombre: Franco Zautzik

Profesor/a: Patricio Poblete

Auxiliares: Gabriel Flores

Sven Reisenegger

Fecha de entrega: 30/04/2018

Introducción:

En esta instancia se buscó resolver la parentización óptima de una multiplicación de matrices. Es decir, dado una multiplicación de matrices de la forma A*B*C*D cuál es la mejor forma de asociar las matrices para minimizar la cantidad de operaciones.

El programa, dado un string con las dimensiones de las matrices a multiplicar, debía devolver un string que representa cuál es la parentización óptima para la multiplicación. Por ejemplo:

2 3 5 4 (multiplicación de matrices A de 2x3, B de 3x5 y C de 5x4)

devuelve

((..).) (que representa la parentización óptima de A*B*C, ((A*B)*C))

Para resolver este conocidísimo problema de programación dinámica se implementó en java un algoritmo de programación dinámica donde se calcula el costo de cada parentización y se elige aquella parentización con mínimo costo.

Diseño de la solución:

Para resolver el problema planteado primero se tuvo que rescatar la información ingresada por el usuario. Pare ello se transformó el string rescatado por Scanner en un arreglo que contuviera sólo las dimensiones.

Teniendo lo anterior y con la con toda la astucia de un programador se buscó dividir *(para reinar)* el problema principal de cómo hacer la primera parentización en subproblemas más pequeños. Si la solución óptima pone el primer paréntesis en la k-ésima matriz luego el problema se reduce a buscar donde posicionar los paréntesis en las sub cadenas de A1 a Ak y de Ak+1 a An.

Es así cómo se genera la recursión que resuelve el problema original, sin embargo el problema de encontrar la posición k-ésima óptima queda aún irresuelto. Luego para determinar esta posición k-ésima se recurre a calcular los costos de ponerla en cada una de las posiciones posibles (k = 1, 2, 3, …, n-1), y luego establecer k como aquel que minimiza los costos.

Luego, en el caso general, se tiene que para obtener k en la cadena Ai a Aj se calcula el costo de posicionarlo en las posiciones k = i, i+1, …, j-1. Luego para obtener el costo mínimo m[i , j] se resuelve el siguiente problema de optimización:

m[i , j] = 0 si i == j

(pues no hay costo de parentizar una matriz consigo misma)

min {m[i , k] + m[k + i , j] + p(i-1)*p(k)*p(j)} si i < j

i <= k < j

Lo anterior es muy costoso con recursión (más que el método de fuerza bruta) a secas así que es aquí cuando entra en acción la programación *dinámica*. El programa recursivo parece olvidar cada vez que resuelve un problema de manera tal que resuelve múltiples veces una misma operación (overlapping problems). La solución al problema anterior es claramente dotar al programa de una “Memoria”, una tabla donde se puedan tabular las soluciones de los subproblemas.

Finalmente una vez establecida la posición verificando el costo mínimo anterior solo hace falta asociar segun el k óptimo.

Implementación:

La implementación del diseño anterior en java es simplemente la traducción código de todo lo ya establecido. Se creó la clase OrdenOptimo en el cual su main recopila la información del usuario, la traduce a un arreglo y luego llama a los métodos que resolvieran el problema (matrizmys) y retornarán la solución ().

El único trozo de código verdaderamente relevante para la solución del problema es el método matrizmys en el cual se lleva a cabo la programación dinámica.

```
//resuelve el problema de orden otimo dando las distintas
posiciones k en la matriz s
    public static int[][] matrizmys(int[] dims){
     //se crean la matriz de costos m y de posiciones para
parentizar s
     int d = dims.length - 1;
        int[][] m = new int[d+1][d+1];
        int[][] s = new int[d+1][d+1];
        //cadenas de tamaño 2 hasta d
        for (int c = 2; c <= d; c++){
          //parte desde la matriz i hasta la matriz j para
resolver los sub problemas
            for (int i = 1; i <= d - c + 1; i++){
                int j = i + c - 1;
                m[i][j] = Integer.MAX_VALUE;
                //costo de posicionar la parentizacion en k
en la sub cadena entre i y j
                for (int k = i; k < j; k++){
                    int cst = m[i][k] + m[k + 1][j] +
dims[i-1]*dims[k]*dims[j];
                    if (cst <= m[i][j]){
                        m[i][j] = cst;
                        s[i][j] = k;
                    }
                }
            }
        }
        //retorna matriz de posiciones k de los distintos
sub problemas
        return s;}
```

Resultados y conclusiones:

Finalmente se llegó al resultado deseado para el programa dinámico. Algunos ejemplos son los siguientes:

1. 2 3 5 4 → ((..).)
2. 1 2 3 4 5 6 → ((((..).).).)
3. 5 5 5 5 5 → ((((..).).).)

(Cosa evidente pues el costo es igual independientemente de la parentización.)

4. 3 1 2 → (..)

(Lo cual hace sentido pues son solo 2 matrices)

5. 2 3 5 4 1 2 → ((.(.(..))).)

Quizá los valores más ilustrativos notables son aquellos que intuitivamente hcen sentido pues dejan en claro que el algoritmo está funcionando

Anexo:

1.

```
package ordenoptimo;

import java.util.Scanner;

public class OrdenOptimo {

    //pide y recupera la informacion de las dimensiones
entregada dentro del arreglo dims
    public static void main(String[] args) {

        System.out.println("Dimensiones de las Matrices?");
        Scanner sc = new Scanner(System.in);

        //este while permite que el programa pueda recibir
multiples consultas
        while (sc.hasNextInt()){
            String si = sc.nextLine();
            String[] sp = si.split(" ");
            int l = sp.length;
            int[] dims = new int[l];

            for(int i = 0; i < l; i++) {
               dims[i] = Integer.parseInt(sp[i]);
            }

            int[][] s = matrizmys(dims);
            parenOpti(1, dims.length - 1, s);  // paren
Opti porfavor xdxdxd
            System.out.println(" ");
        }
        sc.close();
    }

    //resuelve el problema de orden otimo dando las
distintas posiciones k en la matriz s
    public static int[][] matrizmys(int[] dims){

     //se crean la matriz de costos m y de posiciones para
parentizar s
     int d = dims.length - 1;
```

```
        int[][] m = new int[d+1][d+1];
        int[][] s = new int[d+1][d+1];

        //cadenas de tamaño 2 hasta d
        for (int c = 2; c <= d; c++){

          //parte desde la matriz i hasta la matriz j para
resolver los sub problemas
            for (int i = 1; i <= d - c + 1; i++){
                int j = i + c - 1;
                m[i][j] = Integer.MAX_VALUE;

                //costo de posicionar la parentizacion en k
en la sub cadena entre i y j
                for (int k = i; k < j; k++){
                    int cst = m[i][k] + m[k + 1][j] +
dims[i-1]*dims[k]*dims[j];
                    if (cst <= m[i][j]){
                        m[i][j] = cst;
                        s[i][j] = k;
                    }

                }
            }
        }
        //retorna matriz de posiciones k de los distintos
sub problemas
        return s;
    }

     //retorna la parentizacion optima de ((A*B)*C) en el
formato ((..).)
    public static void parenOpti(int i, int j, int[][] s){

        if(i != j){
          System.out.print("("); parenOpti(i, s[i][j], s);
parenOpti(j, s[i][j] + 1, s); System.out.print(")");
        }
        else{
          System.out.print(".");
        }
    }
}
```